Raport Rozproszone Systemy Informatyczne

Serwis REST w WCF, klient HTML+JS

Aleksandra Wolska
Szymon Łopuszyński
Konfiguracja dwumaszynowa

Windows Communication Foundation (WCF) to technologia umożliwiająca tworzenie usług sieciowych (services) w aplikacjach .NET, w tym usług REST (Representational State Transfer).

Usługi REST w WCF korzystają z architektury REST, która jest oparta na idei zasobów (np. obiektów danych, znajdujących się pod unikalnymi adresami URI (Uniform Resource Identifier) oraz na czterech podstawowych operacjach HTTP: GET, POST, PUT i DELETE. W przeciwieństwie do bardziej skomplikowanych protokołów usług sieciowych, takich jak SOAP, usługi REST są bezstanowe i korzystają z prostych formatów danych, takich jak JSON i XML. Bezstanowość polega na tym, że każde żądanie od klienta do serwera musi zawierać wszystkie informacje potrzebne do wykonania żądania. Serwer nie przechowuje żadnych informacji o stanie klienta pomiędzy żądaniami. Architektura REST zakłada podział na klienta i serwer, gdzie klient jest odpowiedzialny za interfejs użytkownika, a serwer za przechowywanie i przetwarzanie danych.

Tworzenie usługi REST w WCF obejmuje następujące kroki:

- Definiowanie interfejsu usługi: Interfejs definiuje metody, które będą dostępne dla konsumentów usługi.
- Implementacja usługi: Usługa jest implementowana jako klasa .NET, która implementuje zdefiniowany wcześniej interfejs.
- Konfiguracja usługi: Usługa musi być skonfigurowana do obsługi protokołu HTTP i formatu danych JSON lub XML.
- Hostowanie usługi: Usługa musi być uruchomiona na serwerze, aby była dostępna dla konsumentów.

Choć WCF umożliwia tworzenie usług REST, warto zauważyć, że technologia ta jest stopniowo zastępowana przez nowsze technologie, takie jak ASP.NET Core, które oferują większą elastyczność i wydajność. Wersja .NET 4.8 jest ostatnią wersją frameworka .NET, która obsługuje WCF.

1. Wymagania
    a. Visual Studio 2022 z pakietem roboczym tworzenia aplikacji ASP.NET i aplikacji internetowych, w tym opcjonalny pakiet “Windows Community Foundation”
    b. Dwie maszyny połączone znajdujące się w tej samej sieci udostępnianej przez hotspot, komputer hostujący posiada otwarte w zaporze porty 8080 i 10000
    c. Import autorskiej biblioteki MyData do wyświetlania danych
2. Definiowanie aplikacji serwisu WCF z kontraktem
    a. Tworzymy nowy projekt RestService
    b. Wybieramy szablon “WCF Service Aplication”
    c. W dalszej konfiguracji wybieramy wersję platformy .NET 4.8
    d. zmieniamy nazwę istniejącego interfejsu iService na IRestService, w którym definiujemy kontrakt serwisu wraz z wszystkimi endpointami i ich formatami zapytań oraz odpowiedzi (Xml/Json), oraz kontraktowy typ danych `Person` osoby w naszej bazie.
3. Implementacja kontraktu
    a. W pliku Service1.svc.cs zamieniamy implementacje Service1 na RestService implementujący nasz kontrakt.
    b. W klasie RestService definiujemy metody z interfejsu do obsługi kolekcji użytkowników, oraz pole typu List<Person>, w której będziemy przechowywać dane użytkowników w pamięci programu, oraz statyczne pole _id będące licznikiem indeksów.
    c. Do metod obsługujących endpointy dodane zostają tworzenie wyjątków WebFaultException z odpowiednimi kodami statusu http, np. 409. Conflict dla próby powielenia adresu email w bazie osób.
    d. Specyfikujemy pojedynczą instancję serwisu dla wszystkich wywołań anotacją `[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]`
    e. Modyfikujemy plik Web.config aby obsłużyć zapytania REST. W sekcji <system.serviceModel> dodajemy opis naszej usługi, a następnie w sekcji <behaviors> specyfikujemy zachowanie naszych endpointów

```
<system.serviceModel>
        <services>
                <service name="MyWebService.RestService">
                        <endpoint address="" binding="webHttpBinding"
contract="MyWebService.IRestService"
behaviorConfiguration="myRESTEndpointBehavior"> </endpoint>
                </service>
        </services>
  <behaviors>
          <endpointBehaviors>
                  <behavior name="myRESTEndpointBehavior">
                          <webHttp helpEnabled="true" />
                  </behavior>
          </endpointBehaviors>
          …
        </services>
</system.serviceModel>
```

Należy upewnić się, że flaga httpGetEnabled w sekcji serviceMetadata ma wartość "true".

4. Włączenie wsparcia dla zapytań CORS
    a. W projekcie tworzymy nowy plik “Global application class” Global.asax. który odpowiada za reagowanie na zdarzenia zgłoszone z poziomu aplikacji i poziomu sesji takie jak start aplikacji, zakończenie sesji, itp. dzięki metodom zaimplementowanym w klasie, reagującym na zdarzenia.
    b. W metodzie Application_BeginRequest, wywoływanej w przypadku otrzymania zapytania. W przypadku odpowiedniej metody dodajemy uprawnienia do headera zapytania

```
protected void Application_BeginRequest(object sender, EventArgs e)
    {

        HttpContext.Current.Response.AddHeader("Access-Control-Allow-Origin", "*");
        if (HttpContext.Current.Request.HttpMethod == "OPTIONS")
        {
            HttpContext.Current.Response.AddHeader(
            "Access-Control-Allow-Methods",
            "POST, PUT, DELETE");

            HttpContext.Current.Response.AddHeader(
            "Access-Control-Allow-Headers",
            "Content-Type, Accept");

            HttpContext.Current.Response.AddHeader(
            "Access-Control-Max-Age", "7200");
            HttpContext.Current.Response.End();
        }
    }
```

    c. W funkcji Application_Start umieszczamy wywołanie funkcji info() z klasy MyData, w celu uzyskania informacji przy uruchomieniu aplikacji.

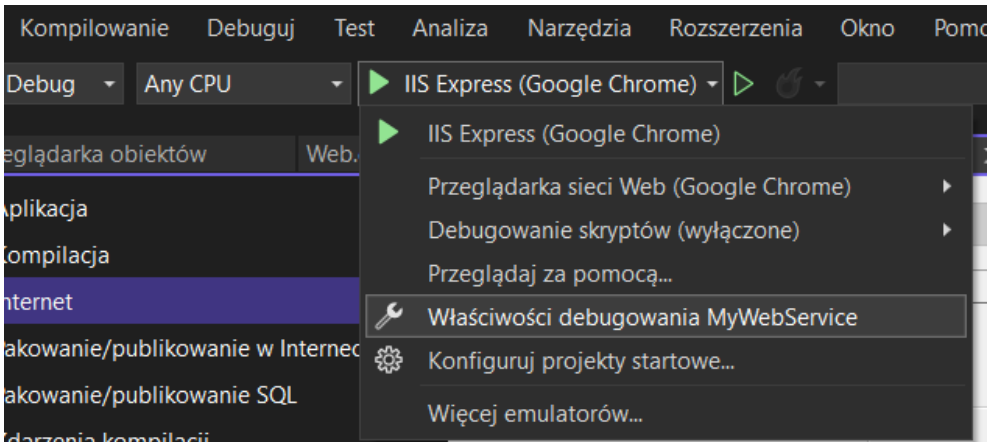
```
protected void Application_Start(object sender, EventArgs e)
    {
        MyData.MyData.info();
    }
```

5. Konfiguracja środowiska dwumaszynowego
    a. Aplikacja serwisu hostowana jest na serwerz IIS Express
    b. W celu konfiguracji środowiska rozproszonego musimy otworzyć porty 8080 i 10000 w zaporze systemowej
    c. Następnie kompilujemy projekt i przechodzimy do pliku splicationhost.config w lokalizacji …\MyWebService\.vs\MyWebService\config
    d. W pliku konfiguracyjnym wyszukujemy znacznik <binding> znajdujący się w sekcji <sites>

e. Dla strony o nazwie takiej jak nazwa naszego projektu w sekcji <bindings> definiujemy dopuszczalne adresy na których serwer IIS może hostować usługę. W naszym przypadku na otwartym porcie 10000 z adresem ip4 komputera 192.168.43.18

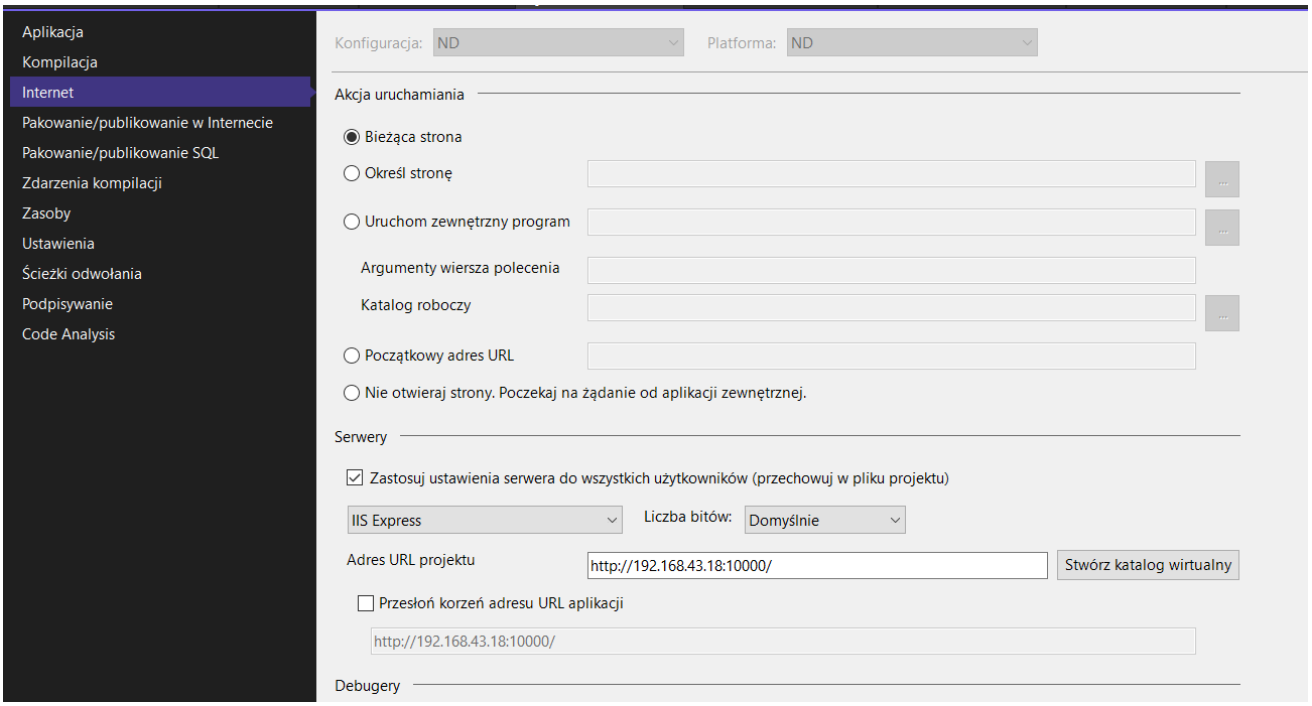
```
<sites>
        <site name="WebSite1" id="1" serverAutoStart="true">
          <application path="/">
            <virtualDirectory path="/" physicalPath="%IIS_SITES_HOME%\WebSite1" />
          </application>
          <bindings>
            <binding protocol="http" bindingInformation=":8080:localhost" />
          </bindings>
        </site>
        <site name="MyWebService" id="2">
          <application path="/" applicationPool="Clr4IntegratedAppPool">
            <virtualDirectory path="/" physicalPath="D:\Studia\Sem6\Rozproszone systemy
informatyczne\Lab7\MyWebService\MyWebService" />
          </application>
          <bindings>
            <binding protocol="http" bindingInformation="*:10000:192.168.43.18" />
          </bindings>
        </site>
        <siteDefaults>
          <!-- To enable logging, please change the below attribute "enabled" to "true" -->
          <logFile logFormat="W3C" directory="%AppData%\Microsoft\IISExpressLogs"
enabled="false" />
          <traceFailedRequestsLogging directory="%AppData%\Microsoft" enabled="false"
maxLogFileSizeKB="1024" />
        </siteDefaults>
        <applicationDefaults applicationPool="Clr4IntegratedAppPool" />
        <virtualDirectoryDefaults allowSubDirConfig="true" />
      </sites>
```

f. Ostatnim krokiem jest ustawienie adresu usługi w konfiguracji serwera IIS.



Wybieramy opcje jak na zdjęciu.

g. Wybieramy zakładkę internet i w sekcji Serwery dla serwera IIS ustawiamy adres na



wcześniej umieszczony w pliku konfiguracyjnym

h. Aplikacje należy uruchamiać z uprawnieniami administratora

Podczas tworzenia klienta wykonane zostały następujące kroki:

1. Ustalanie endpointów i metod HTTP

Zidentyfikowaliśmy potrzebne punkty końcowe (endpoints) i metody HTTP dla klienta. W tym przypadku obsługiwaliśmy endpointy JSON i XML, które obsługują różne metody, takie jak GET, POST, PUT, DELETE. Każdy z tych endpointów jest powiązany z odpowiednim zdarzeniem na przycisku w interfejsie użytkownika.

```
const BASE_URL = "http://192.168.43.18:10000/Service1.svc";
const JSON_ENDPOINT = BASE_URL + "/json"
const XML_ENDPOINT = BASE_URL

document.getElementById('json-get-people-btn').addEventListener('click',
getJsonPeople);
document.getElementById('json-get-person-btn').addEventListener('click',
getJsonPerson);

//inne funkcje

document.getElementById('xml-get-filter-age-btn').addEventListener('click',
getFilteredByAgeXml);
document.getElementById('xml-get-filter-insurance-btn').addEventListener('click
', getFilteredByInsurenceXml);
document.getElementById('xml-get-filter-insurance-class-btn').addEventListener(
'click', getFilteredByInsuranceClassXml);
```

2. Tworzenie funkcji fetchujących

Na podstawie zidentyfikowanych punktów końcowych, stworzyliśmy odpowiednie funkcje fetchujące.
Funkcje fetchujące są to funkcje, które wykonują zapytania do naszego backendu i obsługują odpowiedzi. W naszym przypadku używamy tych funkcji do wykonywania wszystkich operacji CRUD (Create, Read, Update, Delete) na naszym backendzie. Te funkcje korzystają z obiektu XMLHttpRequest, aby komunikować się z naszym backendem i przekazywać odpowiednie dane. XMLHttpRequest jest używany do żądania danych z serwera webowego.
Musieliśmy pamiętać o ustawieniu dobrej konfiguracji dla zapytań do API, aby upewnić się, że żądanie zakończy się sukcesem

- Określenie prawidłowego adresu URL: żądanie musi być skierowane do prawidłowego punktu końcowego API.
- Ustalenie odpowiednich nagłówków: dla zapytań typu POST i PUT, musimy ustawić nagłówek 'Content-Type' na 'application/json' lub 'application/xml' w zależności od używanego formatu.

- Przygotowanie danych do wysłania: w przypadku zapytań typu POST i PUT, musimy przygotować dane do wysłania, które są zgodne ze strukturą oczekiwaną przez API.
- Zdefiniowanie metody HTTP: Musimy prawidłowo ustawić metodę HTTP (GET, POST, PUT, DELETE), której chcemy użyć w naszym zapytaniu.
- Obsługa błędów: W każdym zapytaniu musimy obsłużyć potencjalne błędy, które mogą wystąpić podczas komunikacji z naszym backendem. Na przykład, jeśli status odpowiedzi to coś innego niż 200, to oznacza, że coś poszło nie tak. Funkcja obsługująca błędy wywoływana jest jako callback dla **xhr.onreadyststechange** W przypadku sukcesu zapytania, dane są renderowane w odpowiednim miejscu na stronie. W przypadku błędu, odpowiedni komunikat jest wyświetlany.

XMLHttpRequest to obiekt dostępny w przeglądarce, który pozwala na wysyłanie zapytań HTTP i obsługę odpowiedzi. Pozwala na asynchroniczne zapytania do serwera, co oznacza, że nie blokuje on reszty kodu podczas oczekiwania na odpowiedź.

```
function getJsonPerson() {
    var xhr = new XMLHttpRequest();
    var id = document.getElementById('json-person-id-input').value;
    xhr.onreadystatechange = function () {
        if (this.readyState === 4 && this.status === 200) {
            document.getElementById('json-person-container').innerHTML =
this.responseText;
        } else if (this.readyState === 4) {
            document.getElementById('json-person-container').innerHTML = "An
error occured, data can't be loaded";
        }
    }
    xhr.open("GET", JSON_ENDPOINT + '/people/' + id, true);
    xhr.send();
}
```

```
function postJsonPerson() {
    var xhr = new XMLHttpRequest();
    var name = document.getElementById('json-add-name-input').value;
    var age = document.getElementById('json-add-age-input').value;
    var email = document.getElementById('json-add-email-input').value;
    var insurance =
document.getElementById('json-add-insurance-input').checked;
    var insuranceClass =
document.getElementById('json-add-insurance-class-input').value;
    var data = {
        Name: name,
        Age: age,
        Email: email,
        IsInsured: insurance,
        IncuranceClass: insuranceClass
    };


    xhr.open("POST", JSON_ENDPOINT + '/people', true);
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.send(JSON.stringify(data));

    xhr.onreadystatechange = function () {
        if (this.readyState === 4 && this.status === 200) {
            document.getElementById('json-add-person-container').innerHTML =
"Person added successfully!";
        } else if (this.readyState === 4) {
            document.getElementById('json-add-person-container').innerHTML =
"An error occurred while adding the person.";
        }
    };
}
```

3. Obsługa odpowiedzi

Każda z naszych funkcji fetchujących ma obsługę odpowiedzi, która reaguje na odpowiedź z naszego serwera. Jeśli odpowiedź jest udana (kod statusu HTTP 200), odpowiedź jest prezentowana użytkownikowi. W przypadku błędu (każdy inny kod statusu), prezentowany jest odpowiedni komunikat o błędzie. Ta funkcjonalność polega na wstrzykiwaniu tekstu do wczesniej przygotowanego elementu na stronie

```
index.js
if (this.readyState === 4 && this.status === 200) {
            document.getElementById('json-person-container').innerHTML =
this.responseText;
        } else if (this.readyState === 4) {
            document.getElementById('json-person-container').innerHTML = "An
error occured, data can't be loaded";
        }

index.html
<div id="json-person-container"></div>
```

XMLHttpRequest.readyState jest atrybutem, który zwraca stan XMLHttpRequest:

- 0 - UNSENT: Obiekt został utworzony, ale metoda open() nie została jeszcze wywołana.
- 1 - OPENED: Metoda open() została już wywołana.
- 2 - HEADERS_RECEIVED: Metoda send() została wywołana, a nagłówki i status są dostępne.
- 3 - LOADING: Pobieranie; responseText zawiera częściowe dane.
- 4 - DONE: Operacja zakończona.

4. Przesyłanie danych

Podczas korzystania z metod POST i PUT, musimy przesłać dane do naszego serwera. W tym celu, dane są gromadzone z formularza interfejsu użytkownika, następnie są konwertowane do odpowiedniego formatu (JSON lub XML) i przekazywane do serwera.

```
   xhr.open("POST", JSON_ENDPOINT + '/people', true);
   xhr.setRequestHeader('Content-Type', 'application/json');
   xhr.send(JSON.stringify(data));
```

5. Obsługa różnych typów danych

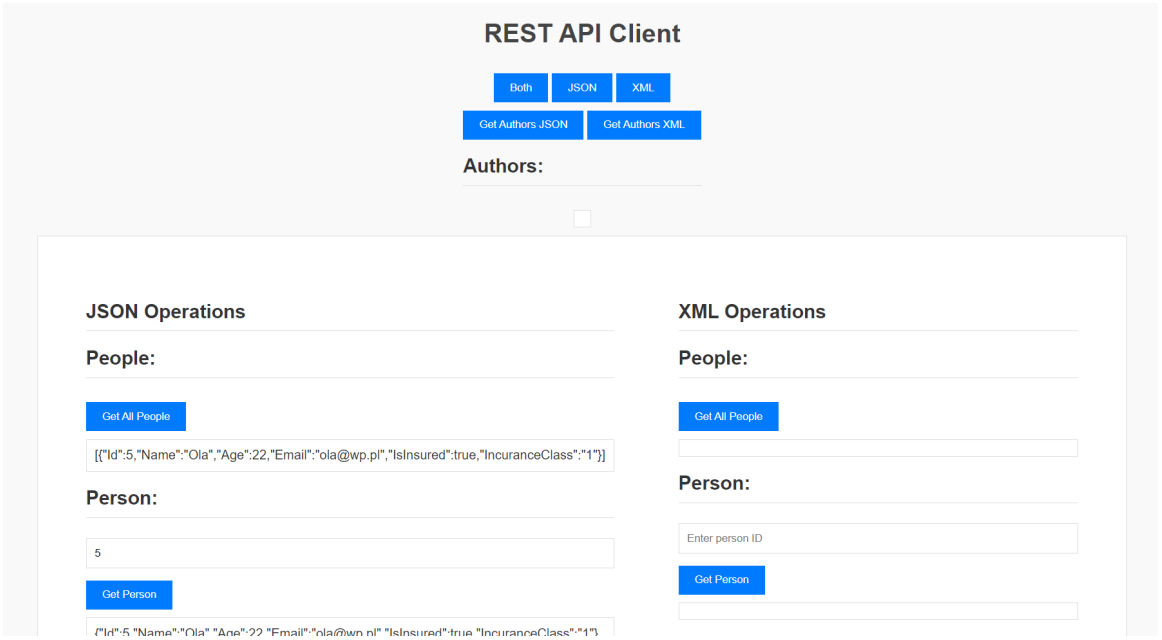
Nasz klient musiał obsłużyć dwa różne typy danych: JSON i XML. Aby to zrobić, stworzyliśmy osobne funkcje fetchujące dla każdego typu danych. Każda z tych funkcji korzysta z odpowiedniego nagłówka 'Content-Type' i konwertuje dane do odpowiedniego formatu przed wysłaniem ich do serwera. W celu zapewnienia odpowiedniego formatu, dla JSON używaliśmy metody JSON.stringify, a dla XML funkcji createXmlData, która tworzy dokument XML z danymi, zapewniając również datacontract.

```
//json
xhr.send(JSON.stringify(data));

//xml

xhr.send(createXmlData(data));

function createXmlData(data) {
    var xmlDoc = document.implementation.createDocument("", "", null);
    var root = xmlDoc.createElement('Person');
    for (var key in data) {
        var element = xmlDoc.createElement(key);
        var text = xmlDoc.createTextNode(data[key]);
        element.appendChild(text);
        root.appendChild(element);
    }
    xmlDoc.appendChild(root);
    return addStringAtIndex(new XMLSerializer().serializeToString(xmlDoc), `
xmlns="http://schemas.datacontract.org/2004/07/MyWebService"
xmlns:i="http://www.w3.org/2001/XMLSchema-instance"`, 7).toString();
}
```

6. Stworzenie interfejsu użytkownika
Podczas projektowania klienta, celem było stworzenie prostego i intuicyjnego interfejsu użytkownika. W tym celu dodaliśmy przyciski, które pozwalają użytkownikowi wybrać, które sekcje chce zobaczyć: JSON, XML lub obie. Za pomocą tych przycisków i funkcji 'toggleJsonSection', 'toggleXmlSection' i 'toggleBothSections', użytkownik ma możliwość wyboru, które sekcje chce wyświetlić. Dzięki temu interfejs jest bardziej przejrzysty i dostosowany do potrzeb użytkownika.

Interfejs pozwala użytkownikowi na wybór rodzaju interakcji z backendem, np. przeglądanie osób, dodawanie nowych osób, aktualizowanie danych istniejących osób oraz usuwanie osób. Użytkownik może również filtrować osoby według wieku, ubezpieczenia i klasy ubezpieczenia.

6. Działanie programu



Dodanie osoby:

Ponowne dodanie osoby z tym samym email:

Update osoby o nie istniejącym indeksie

Usunięcie istniejącej osoby:

Delete Person:

6

Delete Person

Person deleted successfully!

Usunięcie nieistniejącej osoby

Delete Person:

9

Delete Person

An error occurred while deleting the person.

Filtrowanie po wieku:

Filter by Age:

22

Filter

[{"Id":5,"Name":"Ola","Age":22,"Email":"ola@wp.pl","IsInsured":true,"IncuranceClass":"1"}]

Filtrowanie po fakcie bycia ubezpieczonym:

Filter by Insurance:

Insured ☑ Filter

[{"Id":5,"Name":"Ola","Age":22,"Email":"ola@wp.pl","IsInsured":true,"IncuranceClass":"1"}]

Filtrowanie po klasie ubezpieczenia:

Źródła:

- Instrukcja Dr. Frasia - Ćwiczenie 4 - wersja WCF
- Instrukcja Dr. Frasia - Ćwiczenie 5 (wersja c) Ajax-1c-en
- film pokazujący zmianę adresu ip serwera IIS : https://www.youtube.com/watch?v=_t9u9DKIKP4&t=154s&ab_channel=ITProGuide
- Wykład dr. Frasia
- kody statusów http: https://www.websitepulse.com/kb/4xx_http_status_codes